Instruction Format

Introduction

The MIPS R2000/R3000 ISA has fixed-width 32 bit instructions. Fixed-width instructions are common for RISC processors because they make it easy to fetch instructions without having to decode. These instructions must be stored at word-aligned addresses (i.e., addresses divisible by 4).
The MIPS ISA instructions fall into three categories: R-type, I-type, and J-type. Not all ISAs divide their instructions this neatly. This is one reason to study MIPS as a first assembly language. The format is simple.

R-type

R-type instructions refer to register type instructions. Of the three formats, the R-type is the most complex.
This is the format of the R-type instruction, when it is encoded in machine code.

B_{31-26}	B_{25-21}	B_{20-16}	B_{15-11}	B_{10-6}	B_{5-0}
opcode	**register s**	**register t**	**register d**	**shift amount**	**function**

The prototypical R-type instruction is:

```
add $rd, $rs, $rt
```

where $rd refers to some register **d** (**d** is shown as a variable, however, to use the instruction, you must put a number between 0 and 31, inclusive for **d**). **$rs**, **$rt** are also registers.
The semantics of the instruction are;

```
R[d] = R[s] + R[t]
```

where the addition is signed addition.
You will notice that the order of the registers in the instruction is the destination register (**$rd**), followed by the two source registers (**$rs** and **$rt**).

However, the actual binary format (shown in the table above) stores the two source registers first, then the destination register. Thus, how the assembly language programmer uses the instruction, and how the instruction is stored in binary, do not always have to match.

Let's explain each of the fields of the R-type instruction.

- **opcode** (B_{31-26})Opcode is short for "operation code". The opcode is a binary encoding for the instruction. Opcodes are seen in all ISAs. In MIPS, there is an opcode for **add**. The opcode in MIPS ISA is only 6 bits. Ordinarily, this means there are only 64 possible instructions. Even for a RISC ISA, which typically has few instructions, 64 is quite small. For *R-type* instructions, an additional 6 bits are used (B_{5-0}) called the *function*. Thus, the 6 bits of the opcode and the 6 bits of the function specify the kind of instruction for *R-type* instructions.

- **rd** (B_{25-21})This is the *destination register*. The destination register is the register where the result of the operation is stored.

- **rs** (B_{20-16})This is the first *source register*. The source register is the register that holds one of the arguments of the operation.

- **rt** (B_{15-11})This is the second *source register*.

- **shift amount** (B_{10-6})The amount of bits to shift. Used in shift instructions.

- **function** (B_{5-0})An additional 6 bits used to specify the operation, in addition to the opcode.

I-type instructions

I-type is short for "immediate type". The format of an I-type instuction looks like:

B_{31-26}	B_{25-21}	B_{20-16}	B_{15-0}
opcode	**register s**	**register t**	**immediate**

The prototypical I-type instruction looks like:

```
add $rt, $rs, immed
```

In this case, **$rt** is the destination register, and **$rs** is the only source register. It is unusual that **$rd** is not used, and that **$rd** does not appear in bit positions $\mathbf{B_{25-21}}$ for both R-type and I-type instructions. Presumably, the designers of the MIPS ISA had their reasons for not making the destination register at a particular location for R-type and I-type.
The semantics of the **addi** instruction are;

$$R[t] = R[s] + (IR_{15})^{16}\ IR_{15-0}$$

where IR refers to the instruction register, the register where the current instruction is stored.

$(IR_{15})^{16}$ means that bit $\mathbf{B_{15}}$ of the instruction register (which is the sign bit of the immediate value) is repeated 16 times. This is then followed by $IR_{15\text{-}0}$, which is the 16 bits of the immediate value.
Basically, the semantics says to sign-extend the immediate value to 32 bits, add it (using signed addition) to register **R[s]**, and store the result in register **$rt**.

J-type instructions

J-type is short for "jump type". The format of an J-type instuction looks like:

$B_{31\text{-}26}$	$B_{25\text{-}0}$
opcode	**target**

The prototypical I-type instruction looks like:

```
j target
```

The semantics of the **j** instruction (**j** means jump) are:

$$PC <- PC_{31-28}\ IR_{25-0}\ 00$$

where PC is the program counter, which stores the current address of the instruction being executed. You update the PC by using the upper 4 bits of the program counter, followed by the 26 bits of the target (which is the lower 26 bits of the instruction register), followed by two 0's, which creates a 32 bit address. The jump instruction will be explained in more detail in a future set of notes.

Why Five Bits?

If you look at the **R-type** and **I-type** instructions, you will see 5 bits reserved for each register. You might wonder why.
MIPS supports 32 integer registers. To specify each register, the register are identified with a number from 0 to 31. It takes $\mathbf{\log_2 32 = 5}$ bits to specify one of 32 registers.

If MIPS has 64 register, you would need 6 bits to specify the register.

The register number is specified using unsigned binary. Thus, 00000 refers to **$r0** and 11111 refers to register **$r31**.

Why Study Instruction Formats

You might wonder why it's important to study instruction formats. They seem to be arbitrarily constructed. Yet, they aren't. For example, it's quite useful to have the opcode be the same size

and the same location. It's useful to know the exact bits used for the immediate value. This makes decoding much quicker, and the hardware to handle instruction decoding that much simpler.
Furthermore, you begin to realize what information the instructions store. For example, it's not all that obvious that immediate values are stored as part of the instruction for **I-type instructions.**

If you know that, for example, addi does signed addition, then you can also conclude that the immediate value is represented in 2C. Also, to add the immediate value to a 32-bit register value would mean sign-extending the immediate value to 32 bits.

However, not all I-type instructions encode the 16 bit immediate in 2C. For example, addiu (add immediate unsigned) interprets the 16 bits as UB. It zero-extends the immediate and then adds it to the value stored in a 32 bit register.

Three Operand Instructions

Also, notice that the R-type instructions use three operands (i.e., arguments). In earlier, pre-RISC ISAs, memory was expensive, so ISA designers tried to minimize the number of bits used in an instruction. This meant that there were often two, one, or no operands. How did they manage that? Here's an example of an instruction

```
cisc_add $r1, $r2  # R[1] = R[1] + R[2]
```

One way to reduce the total number of operands is to make one operand both a source and a destination register.

Another approach is to use an implicit register.

```
acc_add $r2  # Acc = Acc + R[2]
```

For example, there may be a special register called the *accumulator*. This register is not mentioned explicitly in the instruction. Instead, it is implied by the opcode.
Early personal computers such as the Apple 2, used ISAs with 1 or 2 registers, and those registers were often part of most instructions, thus they didn't have to be specified.

With memory becoming cheaper, and memory access becoming cheaper, it's become easier to devote more bits to an instruction, and to specify three operands instead of two. This makes it more convenient for the assembly language programmer.

Summary

MIPS instructions fall into three categories: R-type, I-type, and J-type. You should know how the bits are laid out (i.e., what the 6 parts of the R-type instruction are, and how many bits in each of the 6 parts). However, it's unnecessary to memorize opcodes.